Alphabet implementation in OCAML

We will build here an implementation of the concept of an alphabet in OCAML, for later use.

This will likely also be our first use of a *functor*.

The relevant files are alphabet.ml[1] and alphabet.mli[2].

We will start by looking at the interface file. The first thing you will see is:

```
module type ElemsType =
  sig
    type elem
    val compare  : elem -> elem -> int
    val allElems : elem list
  end
```

This is a signature for a module. It is the module used as input to our alphabet. Basically, if someone wants to create a new "alphabet" they need to give us a type for the elements, elem, a comparison function compare : elem -> elem -> int, and a list of all the elements allElems. For instance if we wanted an alphabet with just the numbers 0 and 1, then the type would be int, the comparison function would be the normal compare function, and allElems = [0; 1].

Next up we have a signature for the kind of module type our alphabets will have. It is wrapped inside this module type A business, as opposed to just thrown at the "top level", because our alphabets will be sort of like functions, taking an ElemsType as input and returning the appropriate alphabet module. This module type A is exactly the type for that returned module.

```
module type A =
  sig
    type elem
    type t = elem list

    val allElems : elem list
    val epsilon : t
    val empty : t -> bool
    val length : t -> int
    val append : t -> t -> t
    val concat : t list -> t
    val substring : t -> t -> bool
    val prefix : t -> t -> bool
    val suffix : t -> t -> bool
    val allPrefixes : t -> t list
    val allSuffixes : t -> t list
    val allStrings : int -> t list
    val allStringsLeq : int -> t list
    val compare : t -> t -> int
  end
```

[1] ocaml/alphabet.ml
[2] ocaml/alphabet.mli

So let us have a look at this module type: First off it specifies that there are two new value types created, one elem to represent the element type for the alphabet, and one t to represent the type of strings in the alphabet. Notice that we are exposing the type t, we're telling the world exactly how strings are stored. We could have kept it secret and offered a way to go to and from elem list. But in this instance it is essentially in the definition of a string that it is a list of elements, so not much harm is done by exposing it.

After that we have a list values and functions, performing most common tasks for strings. allElems contains a list of all the elements in the alphabet, epsilon is the empty string, empty is a function to test if our string is the empty string, length returns the length of a string, append concatenates two strings, concat concatenates a whole list of strings, substring, prefix and suffix tell us if the first string is a substring/prefix/suffix of the second, allPrefixes and allSuffixes return all the prefixes/suffixes of a list, and allStrings and allStringsLeq return all strings of (up to) a given length. Finally, we have a couple of functions that would split the string on a given prefix/suffix/substring. Since these might fail to do a split, they return option type values.

Let us see what comes after this module type, before we look at implementation.

```
module Make (Elems : ElemsType) : A with type elem = Elems.elem

module MakeInts  (I: sig val allElems: int list end)  : A with type elem = int
module MakeChars (I: sig val allElems: char list end) : A with type elem = char

module Binary  : A with type elem = int
module Decimal : A with type elem = int
module Chars2  : A with type elem = char
module Chars3  : A with type elem = char
```

The first three are what we call *functors*. so Make is a property of the Alphabet module, so we can reach it as Alphabet.Make, and it takes as "argument" a module Elems of type ElemsType. It returns a module of type A, where the meaning of elem in that type is "clarified" to be the type Elems.elem.

So a **functor** is a *parametrized module*, that takes as input another module and returns a more "customized" module. It is in that sense similar to a function, but for modules.

We also offer two more functors, to more easily make alphabets out of lists of integers or characters.

Finally, we construct Five specific alphabets, for binary inputs (0/1), decimal inputs (0-9), and small character sets ('a'/'b', 'a'/'b'/'c', 'a'/'b'/'c'/'d').

Now let's talk implementation, in the .ml file.

First, we need to redeclare the two module types ElemsType and A. Then we build our functor Make, which takes the majority of the file. Let's take a closer look.

Recall that Make has taken as argument another module Elems. So it can use that in its implementations. To start off, it declares the type elem to be Elems.elem. This was something we had to do anyway as the signature for Make told the world we would.

Some of the functions/values that Make is supposed to create are straightforward. allElems comes from Elems.allElems, which we pass through List.sort_uniq to ensure uniqueness. epsilon is just the empty list, empty just tests the "string" for equality with the empty list, length, append and concat all come from the corresponding List methods.

Testing if a string is a prefix of another starts with a pattern match on the two strings. An empty string is always a prefix, and a nonempty string is never a prefix of an empty string. Barring that, the first elements of each string need to match and the remaining substring of the first string needs to be a prefix of the substring of the other.

Testing for suffix is slightly different: To be a suffix a string must either be equal the target string, or else be a suffix of the tail of that string.

To get allPrefixes of a string, we can get all the prefixes of its tail, then prepend the head element to them, and finally include the empty string.

To get allSuffixes of a string we just need to include the string itself to the suffixes of its tail.

For s_1 to be a substring of another string s_2 it must be a prefix of one of the suffixes of s_2. (We could also have described it as a suffix of one of the prefixes)

Next we have three functions to split a string at a specific prefix/suffix/substring, and to return the remaining piece(s). The main theme in all these functions is two pattern matches: One to see if we have arrived at a trivial case (typically end of string or end of substring) and another on the result of a recursive call, to see if it found a match in the "rest".

Lastly, we have the functions that create lists of all strings of a given length. These are a good illustration of List.map together with breaking the problem down to smaller functions.

And of course, we should implement lexicographic ordering, done in the function compare.